Contract-Aware Secure Compilation

Marco Guarnieri
IMDEA Software Institute

Marco Patrignani
Stanford University
CISPA Helmholz Center for Information Security

arXiv:2012.14205v1 [cs.CR] 28 Dec 2020

1 Introduction

Microarchitectural attacks, such as Spectre [8], Meltdown [9], Foreshadow [14], RIDL [15], and ZombieLoad [13], exploit the abstraction gap between the Instruction Set Architecture (ISA) and how instructions are actually executed by processors to compromise the confidentiality and integrity of a system. That is, these attacks exploit microarchitectural side-effects resulting from processor's optimizations, such as speculative and out-of-order execution, and from processor's internal buffers and caches that are invisible at the ISA level.

To secure systems against microarchitectural attacks, programmers need to reason about and *program against* these microarchitectural side-effects. There is, however, no "unique" reference for microarchitectural side-effects. Even for a single manufacturer, processors subtly differ in security-relevant microarchitectural side-effects across generations. For example, the `clflush` instruction for flushing caches behaves differently across generations of Intel processors [16]. As a result, a program might be secure when run on a processor and insecure when run on another processor providing slightly different guarantees. However, we cannot—and should not—expect programmers to manually tailor programs for specific processors and their security guarantees.

Instead, we could rely on compilers (and the secure compilation community), as they can play a prominent role in bridging this gap: compilers should target specific processors microarchitectural security guarantees and they should leverage these guarantees to produce secure code. This will enable decoupling program-level security (say, ensuring that secrets are not leaked under the ISA semantics), which programmers should enforce, and microarchitectural security (say, preventing leaks of secrets due to microarchitectural side-effects), which is the job of the compiler.

To achieve this, we outline the idea of *Contract-Aware Secure COmpilation* (CASCO) where compilers are parametric with respect to a hardware/software security-contract, an abstraction capturing a processor's security guarantees. That is, compilers will automatically leverage the guarantees formalized in the contract to ensure that program-level security properties are preserved at microarchitectural level.

For concreteness, our overview of CASCO builds on a recent formulation of hardware/software contracts [6] that focuses on data confidentiality (and therefore hypersafety properties). We believe that CASCO is more general and it can be applied also to other classes of security properties.

2 Contract-aware secure compilation

The CASCO framework relies on the following elements: ISA, Hardware and Contract languages, the adversary we consider and the notion of contract-aware compilers.

ISA language: We consider an ISA language L with a notion of programs p (comprising both code and data segments) and of architectural program states $\sigma \in \mathsf{AS}$. L is equipped with an *architectural semantics* $\rightarrow : \mathsf{AS} \times \mathsf{AS}$ that models the execution of programs at the architectural level, mapping an architectural state σ to its successor σ'. Assume given $\mathsf{ATR}\,(\mathsf{p})$, a function that denotes the *Architectural TRaces* of p, derived from the sequence of architectural states $\sigma \cdot \ldots \cdot \sigma_n$ that the execution of σ goes through according to $\rightarrow$.

Hardware: The execution of L-programs at the microarchitectural level is formalised with a *hardware semantics* that relies on *hardware states* $\Sigma = \langle \sigma, \mu \rangle \in \mathsf{HS}$. Hardware states consist of an architectural state σ (as before) and a *microarchitectural state* μ, which models the state of components like predictors, caches, and reorder buffers. A hardware semantics $\Rightarrow : \mathsf{HS} \times \mathsf{HS}$ maps hardware states Σ to their successor Σ'.

Adversary: We consider a hardware-level adversary that can observe parts of the microarchitectural state during execution. Given a program p, $\mathsf{HTR}\,(\mathsf{p})$ denotes the *Hardware TRaces* of p, that is, the sequence of hardware observations $\mathcal{A}(\mu_0) \cdot \ldots \cdot \mathcal{A}(\mu_n)$ that the hardware state Σ of p goes through according to $\Rightarrow$. Here, $\mathcal{A}(\mu)$ maps μ to its attacker-visible components (say, the cache metadata).

Contracts: A contract splits the responsibilities for preventing side-channels between software and hardware, and it provides a concise representation of a processor's microarchitectural security guarantees. Following [6], a *contract* c defines: (1) a notion of contract states $\Xi \in CS$ that extend σ with contract-related components, (2) labels $l \in LC$ representing contract-observations, and (3) a labeled semantics $\rightharpoonup : CS \times LC \times CS$. Given a program p, $CTR_c\,(\mathsf{p})$ denotes the *Contract TRace* of p, that is, the sequence $l_1, \cdots, l_n$ of labels that the contract state Ξ of p goes through according to $\rightharpoonup$.

The contract traces of a program $CTR_c\,(\cdot)$ capture which architectural states are guaranteed to be indistinguishable by a hardware attacker on any hardware platform satisfying the contract:

Definition 1 (Hardware satisfies contract [6])**.** A hardware semantics $\mathsf{HTR}\,(\cdot)$ *satisfies a contract* c (denoted $\mathsf{HTR}\,(\cdot) \vdash c$) if, for all programs p and p' that only vary in the data segment, if $CTR_c\,(\mathsf{p}) = CTR_c\,(\mathsf{p}')$, then $\mathsf{HTR}\,(\mathsf{p}) = \mathsf{HTR}\,(\mathsf{p}')$.

Contract-aware compilers: Contract-aware compilers $(\!|\cdot|\!)$ are parametric with respect to a contract $c \in \mathfrak{C}$, which formalizes a processor's security guarantees. The target program $(\!|c, \mathsf{p}|\!)$ depends on the source p and on the contract c.

Contract-aware compilers can be constructed to preserve many security properties (e.g., cryptographic constant-time and absence of speculative leaks), so long as these properties are expressible in the contract semantics (fortunately, this is often the case [6]). Depending on the property of interest, we then choose different secure compilation criteria and instantiate them with the ISA and contract semantics. Proving that a contract-aware compiler upholds such a criterion demonstrates that the criterion is preserved for all contracts in $c \in \mathfrak{C}$, which determine the target language's semantics.

As an example, consider the security property of interest being the prevention of all microarchitectural leaks of information not exposed by ISA observations (captured by the architectural traces $\mathsf{ATR}\,(\cdot)$); this can ensure, for instance, the absence of leaks of transiently accessed data [11]. We therefore choose the secure compilation criterion preserving 2-hypersafety properties [1, 2]. An instantiation of that criterion is found Definition 2 below. That informally tells that the compiler translates ISA-equivalent programs into contract-equivalent ones, so there is no more leakage at the contract level than what expressable in the ISA.

Definition 2 (Compiler satisfies contract)**.** We say that a compiler $(\!|\cdot|\!)$ is secure for all contracts of $\mathfrak{C}$ (denoted as $(\!|\cdot|\!) \vdash \mathfrak{C}$) if for all contracts $c \in \mathfrak{C}$ and programs p, p' that only differ in the data segment, if $\mathsf{ATR}\,(\mathsf{p}) = \mathsf{ATR}\,(\mathsf{p}')$, then $CTR_c\left((\!|c, \mathsf{p}|\!)\right) = CTR_c\left((\!|c, \mathsf{p}'|\!)\right)$.

Theorem 1 illustrates the overarching benefits of using CASCO. It is sufficient to show that both the hardware and the compiler satisfy a contract (Definition 1 and Definition 2) to derive that any ISA program p will produce hardware executions that will not be vulnerable to attacks when run on hardware satisfying the contract. Notably, proofs of Definition 1 and Definition 2 can be done separately and by different parties: hardware developers can provide contracts and proving Definition 1 independently of specific compiler criteria, while developers of secure compilers can focus on proving Definition 2 ignoring most of the hardware details (except those captured by contracts).

Theorem 1. *If* $(\!|\cdot|\!) \vdash \mathfrak{C}$, $c \in \mathfrak{C}$, *and* $\mathsf{HTR}\,(\cdot) \vdash c$, *then for all programs* p *and* p' *that only differ in the data segment, if* $\mathsf{ATR}\,(\mathsf{p}) = \mathsf{ATR}\,(\mathsf{p}')$, *then* $\mathsf{HTR}\left((\!|c, \mathsf{p}|\!)\right) = \mathsf{HTR}\left((\!|c, \mathsf{p}'|\!)\right)$.

3 CASCO for secure speculation

To illustrate the benefits of CASCO, we focus on speculative execution attacks (Spectre) as an example due to the availability of compiler-level countermeasures [11] and security contracts [6]. CASCO, however, is more general and it can be applied to all settings where microarchitectural attacks are prevented by compiler-inserted countermeasures.

Consider the classical Spectre v1 attack [8]. There, an **attacker** poisons the **branch predictor** (which exists at the **hardware** level and not at the ISA) to trigger speculative execution and encode speculatively accessed data (otherwise unaccessible) into the **cache**, so the attacker can later retrieve them by probing the **cache**.

There exist four different *contracts* that serve as specifications of processors' microarchitectural security guarantees [6] and that compilers can use as security specification.

Contract c_{ct}^{seq}: This contract exposes the program counter and the locations of memory accesses on sequential, non-speculative paths. c_{ct}^{seq} is often used to formalize constant-time programming [3, 4], and it is satisfied (in the sense of Definition 1) by in-order, non-speculative processors [6].
Contract c_{ct}^{spec}: This contract additionally exposes the program counter and the locations of all memory accesses on speculatively executed paths [5]. Simple speculative out-of-order processors satisfy c_{ct}^{spec} [6].
Contract c_{arch}^{seq}: This contract, which guarantees the confidentiality of data that is *only transiently* loaded, exposes the program counter, the location of all loads and stores, and the values of all data loaded from memory on standard, i.e., non-speculative, program paths. Processors implementing speculative taint tracking [17, 18] satisfy c_{arch}^{seq} [6].
Contract $c_{ct-pc}^{seq-spec}$: This contract exposes program counter and addresses of loads during sequential execution, and only the program counter during speculative execution. Processors with load-delay countermeasures [12] satisfy $c_{ct-pc}^{seq-spec}$ [6].

A possible countermeasure against Spectre v1 attacks, implemented in the Microsoft Visual C++ and Intel ICC compilers [7, 10], is the insertion of `lfence` instructions (which stop speculation). The countermeasure has been developed to work against speculative, out-of-order processors (contract c_{ct}^{spec}), and it injects an `lfence` instruction after all branch instructions, preventing the attack described before. However, a contract-aware compiler can rely on the contract information to know the underlying processor's security guarantees and optimise its code, avoiding the injection of unnecessary `lfences`. For example, consider processors that implement load-delay (contract $c_{ct-pc}^{seq-spec}$) or speculative taint-tracking countermeasures (contract c_{arch}^{seq}). A contract-aware compiler targeting those processors can avoid inserting `lfences` after branches since there, the speculative memory leaks are prevented by the hardware.

4 Future directions

We believe CASCO provides foundations for designing and proving the correctness of compilers that automatically leverage hardware-level security guarantees, formalized using security contracts, to prevent microarchitectural leaks. For

this, we will need (1) formal languages for modeling interesting classes of contracts; (2) ways of formalizing compilers that use contract information to optimise code; and (3) new proof techniques that account for contract parametricity and composability (to simplify proofs across similar contracts).

Acknowledgements: This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by a grant from Intel Corporation, Juan de la Cierva-Formación grant FJC2018-036513-I, Spanish project RTI2018-102043-B-I00 SCUM, and Madrid regional project S2018/TCS-4339 BLOQUES.

References

[1] Carmine Abate, Roberto Blanco, Stefan Ciobaca, Alexandre Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, , Eric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *ESOP 2020*.

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *CSF 2019*.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'16)*. USENIX Association.

[4] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2019. System-Level Non-interference of Constant-Time Cryptography. Part I: Model. *Journal of Automatic Reasoning* 63, 1 (2019).

[5] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*. IEEE.

[6] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2020. Hardware/Software Contracts for Secure Speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*. IEEE.

[7] Intel. 2018. Using Intel Compilers to Mitigate Speculative Execution Side-Channel Issues. https://software.intel.com/en-us/articles/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues.

[8] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*. IEEE.

[9] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association.

[10] Andrew Pardoe. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/.

[11] Marco Patrignani and Marco Guarnieri. 2019. Exorcising Spectres with Secure Compilers. *CoRR* abs/1910.08607 (2019).

[12] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. ACM.

[13] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. ACM.

[14] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association.

[15] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*. IEEE.

[16] Pepe Vila, Andreas Abel, Marco Guarnieri, Boris Köpf, and Jan Reineke. 2020. Flushgeist: Cache Leaks from Beyond the Flush. *CoRR* abs/2005.13853 (2020).

[17] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. IEEE/ACM.

[18] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*. IEEE/ACM.